Science of Computer Programming 14 (1990) 281-304
North-Holland

TYPES AND INVARIANTS IN THE REFINEMENT CALCULUS*

Carroll MORGAN

Programming Research Group, Oxford University, 8-11 Keble Road, Oxford, OX1 3QD, UK

Trevor VICKERS

Department of Computer Science, Australian National University, Canberra 2601, Australia

Received January 1990

Abstract. A rigorous treatment of types as sets is given for the refinement calculus, a method of imperative program development. It is simple, supports existing practice, casts new light on type-checking, and suggests generalisations that might be of practical benefit. Its use is illustrated by example.

1. Introduction

Program developments in the style of Dijkstra [2] rely on *implicit* typing of variables. One agrees beforehand that all variables have a certain type (say $\mathbb{Z}$, the integers); then individual steps are justified by referring to that type, where necessary. For example, the truth of the entailment

$$(a<b) \Rrightarrow wp(a := a+1, a \leq b)$$

depends on a and b being integers. But that dependence is at present informal.

In the refinement calculus also [1, 12, 14], the dependence is informal; and the contribution of this paper is to make it rigorous. We make *typed* local variable declarations affect the meaning of commands within their scope, and allow development steps there to refer to that type information.

In fact, typing is a special kind of invariant: in the scope of the declaration **var** $n:\mathbb{N}$, which introduces a new local variable n of type $\mathbb{N}$ (the natural numbers), the invariant is $n \in \mathbb{N}$ and all commands preserve it. We allow the declaration of *local invariants* in general, and the rules for typing follow from that.

A surprising feature of our approach is that imposing an invariant does not increase the developer's proof obligations in the usual (prohibitive) way: it is not necessary to prove, during development, that the invariant is maintained. It is maintained automatically.

* An expansion of [10].

Thus even programs that appear to break the invariant actually maintain it: instead of being type-incorrect, they are miracles [9, 14, 16]. But miracles are still programs, and that allows a more uniform calculus of program refinement.

Nevertheless, a check is necessary to exclude miracles from the final program, since they cannot be executed. That check, like the type-checking which it subsumes, is often obvious and can in many cases be delegated to machine. When a machine cannot perform it, it is only because the program developer has used more general invariants than typing.

We believe that it is important to separate the use of an invariant (or type) from the proof that it is respected. Continual formal type-checking *during* development is impractical—and that has, so far, limited the rigorous use of types in imperative programs.

This paper extends the earlier [10] with the examples of Section 9.

2. Invariant semantics

We retain Dijkstra's language, but now give its meaning relative to an invariant, which we call the *context*. Any formula over the program variables (even *false*) may be a context. We write $wp_I(P, \phi)$ for the weakest precondition in context I of a program P with respect to a postcondition ϕ, and give the resulting semantics of Dijkstra's language in Fig. 1. Note that taking I to be *true* in Fig. 1 gives the usual semantics: therefore we say that *true* is the default context.

The following lemmas support our choice of semantics in Fig. 1.

$$
\begin{aligned}
wp_I(\textbf{skip}, \phi) &\triangleq I \wedge \phi \\
wp_I(\textbf{abort}, \phi) &\triangleq \mathit{false} \\
wp_I(x := E, \phi) &\triangleq I \wedge (I \Rightarrow \phi)[x \backslash E] \\
wp_I(P; Q, \phi) &\triangleq wp_I(P, wp_I(Q, \phi)) \\
wp_I(\textbf{if } (\Box i \bullet G_i \rightarrow P_i) \textbf{ fi}, \phi) &\triangleq (\bigvee i \bullet G_i) \wedge \\
&\quad\ (\bigwedge i \bullet G_i \Rightarrow wp_I(P_i, \phi))
\end{aligned}
$$

The substitution $[x \backslash E]$ replaces all free occurrences of x by E, with suitable renaming of bound variables if necessary to avoid capture. Iteration **do** ... **od**, a special case of recursion, is dealt with in Section 8.3.

Fig. 1. Invariant semantics for Dijkstra's language.

Lemma 2.1 (Assume invariant). *No program P, in context I, is guaranteed to terminate unless I holds initially*:

$$wp_I(P, \mathit{true}) \Rrightarrow I.$$

Proof. Structural induction over P. □

Lemma 2.2 (Establish invariant). *Any program P, in context I, establishes I iff it establishes anything*:

$$wp_I(P, \phi) \equiv wp_I(P, I \wedge \phi).$$

Proof. Structural induction over P. □

Note that $wp_I(P, \cdot)$ is still monotonic over implication (in its second argument), and still distributes conjunction. Note also that, in the (typing) context $n \in \mathbb{N}$, the command $n := -1$ terminates, and hence (Lemma 2.2) re-establishes the invariant! We return to that later.

3. The refinement calculus

The refinement calculus is based on an extended programming language, in which specifications can be written, and a relation of refinement between its programs such that implementations refine their specifications [1, 12, 14].

3.1. Language extensions

A *specification* is a list w of changing variables, called the *frame*, and a formula *post*, called the *postcondition*. It is defined as follows:

Definition 3.1 (Specification).

$$wp_I(w:[post], \phi) \triangleq I \wedge (\forall w \bullet I \wedge post \Rightarrow \phi).$$

Quantifications are written within parentheses (...), and the bound variable list is terminated by •. Our precedence for propositional connectives is (highest) $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ (lowest).

In the special case of a specification with an empty list of changing variables, we have a *coercion*, whose definition is derived from Definition 3.1:

Definition 3.2 (Coercion).

$$wp_I([post], \phi) \triangleq I \wedge (post \Rightarrow \phi).$$

An *assertion* is a single condition *pre*, written $\{pre\}$, and is defined as follows:

Definition 3.3 (Assertion).

$$wp_I(\{pre\}, \phi) \triangleq I \wedge pre \wedge \phi.$$

Coercions and assertions are together known as *annotations*. We omit the sequential composition operator ; whenever one or both of its arguments is an annotation.

Assertions and specifications often occur together; for example, the program

$$\{m > 0\}\ n:[n < m] \tag{1}$$

sets n below m provided m was positive to begin with. (Note that Definitions 3.1 and 3.3 differ slightly from the notation of [9], where specification (1) would be written $n:[m > 0, n < m]$. The present notation agrees with [14]; but [1] remains substantially different.)

An untyped local variable x is introduced using the declaration **var** and scope brackets $|[\ldots]|$. It is defined as follows:

Definition 3.4 (Untyped local variable). Provided neither I nor ϕ contains free x,

$$wp_I(|[\mathbf{var}\ x \bullet P]|, \phi) \ \hat{=}\ (\forall x \bullet wp_I(P, \phi)).$$

Definition 3.4 is standard; later, we extend it for typed local variables.

A local invariant J is introduced by the declaration **inv** and scope brackets. It is defined as follows:

Definition 3.5 (Local invariant).

$$wp_I(|[\mathbf{inv}\ J \bullet P]|, \phi) \ \hat{=}\ wp_{I \wedge J}(P, \phi).$$

In $|[\mathbf{inv}\ J \bullet P]|$, the new invariant J is assumed initially, is maintained automatically by every command in P, and therefore is established finally.

Finally, typed local variables are a combination of the above: an untyped declaration, an initialisation, and a local invariant. We have:

Definition 3.6 (Typed local variables). For any set T and formula I:

$$\begin{array}{l} |[\mathbf{var}\ x : T\ \mathbf{and}\ I \bullet P]| \\ \hat{=} |[\mathbf{var}\ x \bullet x : [x \in T \wedge I]; \\ \qquad\qquad |[\mathbf{inv}\ x \in T \wedge I \bullet P]| \\ ]|. \end{array}$$

The "$: T$" gives the type of x; the "**and** I" gives an invariant to be imposed additionally. An invariant *true* may be omitted.

Like **inv**, the declaration **and** may appear on its own, so that

$$|[\mathbf{var}\ x : T\ \mathbf{and}\ I \bullet \ldots]|$$

is equivalent to the nested declarations

$$|[\mathbf{var}\ x : T \bullet |[\mathbf{and}\ I \bullet \ldots]|\,]|.$$

The difference between **inv** and **and** is only that the latter includes an initialisation.

Invariants introduced by **inv** or **and** are *explicit*; invariants introduced by typing $x : T$ are *implicit*.

3.2. The refinement relation

The refinement relation $\sqsubseteq$ holds between programs P and Q whenever Q satisfies every specification that P does, in every context. This is its definition:

Definition 3.7 (Refinement). For programs P and Q, we have $P \sqsubseteq Q$ iff for all contexts I and postconditions ϕ,

$$wp_I(P, \phi) \Rrightarrow wp_I(Q, \phi).$$

Examples of refinement are given in the laws of Section 4.

If a program fragment is contained within the scope of a local invariant I, we can take advantage of the invariant by exploiting a weaker refinement relation $\sqsubseteq_I$ defined as follows:

Definition 3.8 (Refinement in context). For programs P and Q:

$$(P \sqsubseteq_I Q) \;\hat{=}\; |[\textbf{inv } I \bullet P]| \sqsubseteq |[\textbf{inv } I \bullet Q]|.$$

In Definition 3.8, the index I of $\sqsubseteq_I$ is the context in which the refinement is valid. For example, with $I \hat{=} (n \in \mathbb{N})$, we have

$$x:[x \geq 0] \sqsubseteq_I x := n.$$

That is, setting x to a natural number n refines setting it to any nonnegative value. (Law 4.4 below supplies an easy proof.)

In practice, we use the following lemma to demonstrate refinements in context:

Lemma 3.9 (Refinement in context). *For programs P and Q, we have $P \sqsubseteq_I Q$ iff for all postconditions ϕ, and stronger contexts J with $J \Rrightarrow I$,*

$$wp_J(P, \phi) \Rrightarrow wp_J(Q, \phi).$$

Proof. From Definitions 3.8, 3.7 and 3.5, $P \sqsubseteq_I Q$ iff for all postconditions ϕ and contexts K:

$$wp_{I \wedge K}(P, \phi) \Rrightarrow wp_{I \wedge K}(Q, \phi).$$

But the set of contexts "$I \wedge K$ for all K" is exactly the set of contexts "J with $J \Rrightarrow I$", as required. □

An immediate consequence of Lemma 3.9 is that strengthening the context cannot invalidate a refinement:

Lemma 3.10 (Strengthen context). *If $P \sqsubseteq_I Q$ and $J \Rrightarrow I$, then also $P \sqsubseteq_J Q$.*

Proof. Trivial, since Lemma 3.9 treats all stronger J than I. □

4. A development method

We regard as our "programming" language all the constructions of Fig. 1 (traditional) and Section 3.1 (novel). That includes abstract programs, like

$$x:[x^2 - 3x + 2 = 0],$$

and it of course admits "ordinary" programs too, like $x := 1$ and $x := 2$ (both of which refine the above).

Though "programming" and "ordinary" we use informally, we do precisely identify a subset of the language, called *code*, that can be executed automatically by computer. It includes all of Fig. 1, and Definitions 3.3 and 3.6 without **and** of Section 3.1. (But in the case of Definition 3.6, we do restrict the language—in code—with which the set T can be expressed.) Code does *not* include specifications, coercions, untyped local variables, or explicit invariants.

In the refinement calculus, the aim of development is to refine a given program into another one written entirely in code. For that we use refinement laws, several of which are shown in this section. We need not use *wp* directly—it is used only to prove the refinement laws themselves.

For example, the following three laws deal with the use of context for simplification, and are easy consequences of Definition 3.7:

Law 4.1 (Weaken assumption). *Provided* $(I \wedge pre) \Rightarrow pre'$,

$$\{pre\} \sqsubseteq_I \{pre'\}.$$

Law 4.2 (Strengthen postcondition). *Provided* $(I \wedge post') \Rightarrow post$,

$$w:[post] \sqsubseteq_I w:[post'].$$

Law 4.3 (Annotations and **skip**).

$$\{pre\} \sqsubseteq_I \textbf{skip} \sqsubseteq_I [post].$$

Law 4.3 means, for example, that at any point an assumption may be removed or a coercion inserted. But the above laws do not directly make progress towards code—which is after all our overall goal.

There are two defining aspects of code: first, any program written in code must be executable; and second, it must be decidable whether or not any text *is* code. The first aspect is straightforward: the language of code was *designed* for execution. Specifications, however, were not—and, in general, they *are* not, because most of the types of interest do not have recursively axiomatisable theories. Assertions, however, are code because they are refined by **skip** (Law 4.3).

The second aspect of code is straightforward, too, but requires "type-checking". Consider this program:

$$|[\textbf{var } n:\mathbb{N} \bullet n := -1]|. \tag{2}$$

Definitions 3.6, 3.4, 3.1 and 3.5 show that it does terminate in the default context *true*. But it establishes *false*:

$$\begin{aligned}
&wp(|[\textbf{var } n:\mathbb{N} \bullet n := -1]|, false)\\
\equiv\ &(\forall n \bullet wp(n:[n \in \mathbb{N}], wp_{n\in\mathbb{N}}(n := -1, false)))\\
\equiv\ &(\forall n \bullet (\forall n \bullet n \in \mathbb{N} \Rightarrow n \in \mathbb{N} \wedge (-1 \in \mathbb{N} \Rightarrow false)))\\
\equiv\ &true.
\end{aligned}$$

Thus program (2) violates Dijkstra's Law of the Excluded Miracle [2, p. 18], and it cannot, therefore, be code.

That program is not code because it is ill-typed; but we defer the recognition of code to Section 7. Assuming we *can* recognise code, the development method is this: given a program, find code that refines it. Instrumental in that process are laws of refinement that, like the following, introduce code:

Law 4.4 (Assignment). *Provided* $(I \wedge pre) \Rrightarrow post[w \backslash E]$,

$$\{pre\}\ w, x : [post] \sqsubseteq_I \{pre\}\ w := E.$$

Proof. We use Lemma 3.9: suppose $J \Rrightarrow I$. Then

$$\begin{aligned}
& wp_J(\{pre\}\ w, x : [post], \phi) \\
\equiv\ & J \wedge pre \wedge J \wedge (\forall w, x \bullet J \wedge post \Rightarrow \phi) \\
\Rrightarrow\ & J \wedge pre \wedge (J[w \backslash E] \wedge post[w \backslash E] \Rightarrow \phi[w \backslash E]) \\
\Rrightarrow\ & \text{"assumptions"} \\
& J \wedge pre \wedge (J \Rightarrow \phi)[w \backslash E] \\
\equiv\ & wp_J(\{pre\}\ w := E, \phi). \qquad \square
\end{aligned}$$

The variables x in the frame, if any, are those that the assignment declines to change. The assumption $\{pre\}$ may be removed with Law 4.3, leaving only the assignment; but in Section 9 we see that sometimes it is best to leave the assumption there.

An important feature of Law 4.4 is that the context I plays a *constructive* role: the stronger it is, the more likely is the refinement to be valid. That is an example of Lemma 3.10, and is an important practical point: one need not examine *all* invariant declarations in order to apply a particular law.

Also, Law 4.4 illustrates the general coding process: it replaces a non-executable construct, the specification, with code. Similar laws for the remaining constructs (but without invariants) can be found in [1, 12, 14]; the development method itself is the subject of [11].

5. Laws for local invariants

Local invariants are introduced with this law, whose proof uses Lemma 5.2 following.

Law 5.1 (Local invariant). *For any I, J and P,*

$$\{J\}\ P\ [J] \sqsubseteq_I |[\mathbf{inv}\ J \bullet P]|.$$

Proof. Suppose $K \Rrightarrow I$. Then

$$
\begin{array}{ll}
 & wp_K(\{J\}\ P\ [J], \phi) \\
\equiv & J \wedge K \wedge wp_K(P\ [J], \phi) \\
\equiv & J \wedge wp_K(P, K \wedge (J \Rightarrow \phi)) \\
\equiv & \quad \text{"Lemma 2.2"} \\
 & J \wedge wp_K(P, J \Rightarrow \phi) \\
\Rrightarrow & \quad \text{"Lemma 5.2"} \\
 & wp_{K \wedge J}(P, \phi) \\
\equiv & wp_K(|[\textbf{inv}\ J \bullet P]|, \phi). \qquad \square
\end{array}
$$

Lemma 5.2 (Maintain invariant).

$$J \wedge wp_I(P, J \Rightarrow \phi) \Rrightarrow wp_{I \wedge J}(P, \phi).$$

Proof. Structural induction over P; in fact equivalence $\equiv$ holds in every case except sequential composition. $\square$

Note that in Law 5.1 the left-hand side assumes J before P and establishes it after P; the right-hand side maintains it within P as well.

Local invariants can also be introduced implicitly, within typed local variable declarations:

Law 5.3 (Introduce local block). *Provided x is a fresh local variable, not occurring in T or post,*

$$
\begin{array}{ll}
 & w:[post] \\
\sqsubseteq_I & |[\textbf{var}\ x : T\ \textbf{and}\ J \bullet w, x:[post]]|.
\end{array}
$$

Proof. Let $K \Rrightarrow I$, and assume x is a fresh variable. We introduce the abbreviation Φ for the formula $x \in T \wedge J$, and proceed

$$
\begin{array}{ll}
 & wp_K(w:[post], \phi) \\
\equiv & K \wedge (\forall w \bullet K \wedge post \Rightarrow \phi) \\
\equiv & \quad \text{"}x \text{ is fresh"} \\
 & (\forall x \bullet K \wedge (\forall w, x \bullet K \wedge post \Rightarrow \phi)) \\
\Rrightarrow & (\forall x \bullet K \wedge (\forall x \bullet K \wedge \Phi \Rightarrow \\
 & \qquad\qquad K \wedge \Phi \wedge (\forall w, x \bullet K \wedge \Phi \wedge post \Rightarrow \phi))) \\
\equiv & (\forall x \bullet wp_K(x:[\Phi], wp_{K \wedge \Phi}(w, x:[post], \phi))) \\
\equiv & \quad \text{"Definition 3.6"} \\
 & wp_K(|[\textbf{var}\ x : T\ \textbf{and}\ J \bullet w, x:[post]]|, \phi). \qquad \square
\end{array}
$$

Theorem 5.4 below justifies *stepwise refinement* in context: refining a part of a program in context I refines the whole program in that context:

Theorem 5.4 (Monotonicity). *Let $\mathscr{C}$ be a program scheme containing the program name p, and let $\mathscr{C}(X)$ be the result of replacing all occurrences of p in $\mathscr{C}$ by the program X. Then for any programs P and Q, if $P \sqsubseteq_I Q$, then also $\mathscr{C}(P) \sqsubseteq_I \mathscr{C}(Q)$.*

Proof. Structural induction over $\mathscr{C}$. The only novel case is local invariants: suppose $P \sqsubseteq_I Q$. If $J \Rrightarrow I$, then

$$
\begin{aligned}
& wp_J(|[\textbf{inv } K \bullet P]|, \phi) \\
\equiv\ & wp_{J \wedge K}(P, \phi) \\
\Rrightarrow\ & \text{"Lemma 3.9 and assumption, since } J \wedge K \Rrightarrow I\text{"} \\
& wp_{J \wedge K}(Q, \phi) \\
\equiv\ & wp_J(|[\textbf{inv } K \bullet Q]|, \phi). \qquad \square
\end{aligned}
$$

Theorem 5.5 improves Theorem 5.4, and it is *the* reason for introducing a local invariant, since within its scope we can use the refinement relation $\sqsubseteq_{I \wedge J}$, easier to establish than $\sqsubseteq_I$.

Theorem 5.5 (Use local invariant). *Let $\mathscr{C}$ be as before. If $P \sqsubseteq_{I \wedge J} Q$, then*

$$|[\textbf{inv } J \bullet \mathscr{C}(P)]| \sqsubseteq_I |[\textbf{inv } J \bullet \mathscr{C}(Q)]|.$$

Proof. From Theorem 5.4, $\mathscr{C}(P) \sqsubseteq_{I \wedge J} \mathscr{C}(Q)$. The result follows from Definitions 3.8 and 3.5. $\square$

Note that the stronger hypothesis $P \sqsubseteq_I Q$ would be enough in Theorem 5.5: we may use the new invariant J, but are not obliged to do so.

Although local invariants make refinement easier, there is a price to pay later. After the refinement, still the **inv** remains—and it is not code. The next section deals with its elimination.

6. Eliminating local invariants

An implicit invariant, introduced by Law 5.3, need not be eliminated—because in that special case it *is* code. But explicit invariants (Law 5.1, or **and** in Law 5.3) must be removed, and that requires laws like these:

Law 6.1 (Invariant distribution through sequential composition). *For any context J,*

$$
\begin{aligned}
& |[\textbf{inv } I \bullet P; Q]| \\
\sqsubseteq_J\ & |[\textbf{inv } I \bullet P]|; |[\textbf{inv } I \bullet Q]|.
\end{aligned}
$$

Proof. Direct from Definition 3.5 and Fig. 1. (In fact, the two programs are equal.) □

Law 6.2 (Invariant distribution through alternation). *For any context J, suppose that $(J \wedge I) \Rrightarrow (G_i \Leftrightarrow G_i')$ for each branch i of the alternation. Then*

$$\begin{array}{l} \quad |[\textbf{inv } I \bullet \textbf{if } (\Box i \bullet G_i \to P_i) \textbf{ fi}]| \\ \sqsubseteq_J \textbf{ if } (\Box i \bullet G_i' \to |[\textbf{inv } I \bullet P_i]|) \textbf{ fi}. \end{array}$$

Proof. Let $K \Rrightarrow J$. Then

$$\begin{array}{ll} & wp_K(lhs, \phi) \\ \equiv & (\bigvee i \bullet G_i) \wedge (\bigwedge i \bullet G_i \Rightarrow wp_{I \wedge K}(P_i, \phi)) \\ \Rrightarrow & \text{"assumptions, Lemma 2.1"} \\ & (\bigvee i \bullet G_i') \wedge (\bigwedge i \bullet G_i' \Rightarrow wp_{I \wedge K}(P_i, \phi)) \\ \equiv & wp_K(rhs, \phi). \quad \Box \end{array}$$

Law 6.3 (Invariant elimination for assignment). *For any context J, provided $(J \wedge I) \Rrightarrow I[w \backslash E]$,*

$$|[\textbf{inv } I \bullet w := E]| \sqsubseteq_J w := E.$$

Proof. Let $K \Rrightarrow J$. Then

$$\begin{array}{ll} & wp_K(|[\textbf{inv } I \bullet w := E]|, \phi) \\ \equiv & wp_{I \wedge K}(w := E, \phi) \\ \equiv & I \wedge K \wedge (I \wedge K \Rightarrow \phi)[w \backslash E] \\ \Rrightarrow & \text{"assumptions"} \\ & K \wedge (K \Rightarrow \phi)[w \backslash E] \\ \equiv & wp_K(w := E, \phi). \quad \Box \end{array}$$

Law 6.4 (Invariant elimination for **skip**). *For any context J,*

$$|[\textbf{inv } I \bullet \textbf{skip}]| \sqsubseteq_J \textbf{skip}.$$

Proof. Trivial. □

We see in Section 8.2 a law that distributes **inv** over recursion; and there are analogs of Laws 6.3 and 6.4 that deal with **abort**, annotations, and specifications. All of those, together, allow **inv** to be eliminated by first distributing it towards the atomic statements of a program, then discharging certain proof obligations at each separately. That last step, of which Law 6.3 is an example, is like type-checking, to which finally we turn.

7. Type-checking

Type-checking is the automated application, say by a compiler, of the procedure described in Section 6—but in the special case of implicit invariants for typed local variables.

If the types can be made only in certain ways (for example, enumeration, Cartesian product, disjoint union, etc.), and the expressions E in assignments are restricted similarly, then it is decidable whether an expression E is of type T given that we know the types of the constituents of E. Consider, for example, the typing context

$$a,b,c \in \mathbb{Z}, \tag{3}$$

and the assignment $a := b+c$. For the elimination of the invariant (3), we require by Law 6.3:

$$a,b,c \in \mathbb{Z} \;\Rrightarrow\; b+c,b,c \in \mathbb{Z}.$$

And that follows from $(b,c \in \mathbb{Z}) \Rrightarrow (b+c \in \mathbb{Z})$, which can be built in to a compiler.

But now consider a more interesting case: let the invariant be $I \mathrel{\widehat{=}} (m,n \in \mathbb{N})$. By Law 4.4 (taking *pre* to be *true*), we have

$$n:[n = m-1] \sqsubseteq_I n := m-1.$$

And so by Definition 3.6 and Theorems 5.5 and 5.4, we have

$$\begin{aligned} &|[\textbf{var } m, n:\mathbb{N} \bullet n:[n = m-1]]| \\ \sqsubseteq &|[\textbf{var } m, n:\mathbb{N} \bullet n := m-1]|. \end{aligned}$$

To the experienced programmer, that looks unlikely: if the value of m is zero, surely the assignment will abort—yet the specification does not abort! And we are not saved either by (decidable) type checking, unless *all* such assignments are ill-typed: a type-checker cannot know the actual value of m.

In fact, all such assignments *are* ill-typed. Natural number subtraction is not integer subtraction: it differs exactly in the case where the result would be negative. Using $\ominus$ for natural number subtraction, the assignment $n := m \ominus 1$ is well-typed, but the earlier refinement fails:

$$n:[n = m-1] \not\sqsubseteq_I n := m \ominus 1.$$

And that is where it should fail.

Suppose, then, that we know m is positive. In that case we must show

$$\{m > 0\}\, n:[n = m-1] \sqsubseteq_I n := m \ominus 1,$$

and that follows from Law 4.4 provided

$$I \wedge (m > 0) \;\Rrightarrow\; m-1 = m \ominus 1.$$

The proviso is clearly true.

Fortunately, most operators can still be overloaded; we need not distinguish natural and integer addition, for example. And some of the distinctions are already in widespread use: compare / and **div**.

Note further that Definition 3.6 introduces an initialising specification. That can be refined to an assignment, provided the type is non-empty—which, therefore, we require.

Thus, within the above constraints, we can view type-checking as the elimination of a specific kind of local invariant, and it can be done automatically by a compiler.

8. Recursion

8.1. Syntax and semantics

A recursive program is written

$$\textbf{mu}\ p \bullet \mathscr{C}\ \textbf{um}, \tag{4}$$

where p is a program name and $\mathscr{C}$ a program scheme probably containing p. For its meaning, we must understand $\mathscr{C}$ as a function from programs to programs: the application of $\mathscr{C}$ to the program X is just $\mathscr{C}(X)$, the program left when p is replaced in $\mathscr{C}$ by X. The meaning of (4) is then the least fixed-point of that function.

It is not the meanings of programs that must be monotonic for such least fixed-points to exist; and indeed those meanings are not monotonic in their context argument. For example, from Fig. 1 we have

$$wp_{false}(x := 0, x \neq 0) \equiv \mathit{false},$$

$$wp_{x \neq 0}(x := 0, x \neq 0) \equiv \mathit{true},$$

$$wp_{true}(x := 0, x \neq 0) \equiv \mathit{false}.$$

It is the program constructors that must be monotonic over programs; and their monotonicity is stated in Theorem 5.4. (The monotonicity of **mu** ... **um** follows from the monotonicity of *fix*, the least fixed-point operator itself.)

8.2. Eliminating local invariants

We must extend Section 6 with a law that allows **inv** to be eliminated when it surrounds a recursion. This is the law:

Law 8.1 (Invariant elimination for recursion). *Let $\mathscr{C}$ and $\mathscr{D}$ be two program schemes, and for any program X let $\mathscr{C}(X)$ and $\mathscr{D}(X)$ be the programs resulting when the program name p is replaced by X. Suppose that for any program X,*

$$|[\textbf{inv}\ I \bullet \mathscr{C}(X)]| \sqsubseteq_J \mathscr{D}(|[\textbf{inv}\ I \bullet X]|).$$

Then we may eliminate a surrounding invariant as follows:

$$|[\textbf{inv } I \bullet \textbf{mu } p \bullet \mathscr{C}\ \textbf{um}]| \sqsubseteq_J \textbf{mu } p \bullet \mathscr{D}\ \textbf{um}.$$

Proof. Suppose $K \Rrightarrow J$. Then

$$\begin{aligned} & wp_K(|[\textbf{inv } I \bullet \textbf{mu } p \bullet \mathscr{C}\ \textbf{um}]|, \phi) \\ \equiv\ & wp_{K \wedge I}(\textbf{mu } p \bullet \mathscr{C}\ \textbf{um}, \phi) \\ \equiv\ & \mathit{fix}\ \mathscr{C}\ (K \wedge I)\ \phi, \end{aligned}$$

where we use $\mathscr{C}$ also for the *meaning* of the program scheme. (A more exact treatment would use semantic functions with environments, in the style of [19].) Since $\mathscr{C}$ is monotonic, we can continue

$$\equiv \quad \text{"for some ordinal } \beta\text{"}$$

$$\left(\bigvee_{\alpha<\beta} \mathscr{C}^{\alpha}\ \textbf{abort}\ (K \wedge I)\phi\right)$$

$$\equiv \left(\bigvee_{\alpha<\beta} |[\textbf{inv } I \bullet \mathscr{C}^{\alpha}\ \textbf{abort}]|\ K\phi\right)$$

$$\Rrightarrow \quad \text{"assumption, and transfinite induction over } \alpha\text{"}$$

$$\left(\bigvee_{\alpha<\beta} \mathscr{D}^{\alpha}|[\textbf{inv } I \bullet \textbf{abort}]|\ K\phi\right)$$

$$\equiv \left(\bigvee_{\alpha<\beta} \mathscr{D}^{\alpha}\ \textbf{abort}\ K\phi\right)$$

$$\Rrightarrow wp_K(\textbf{mu } p \bullet \mathscr{D}\ \textbf{um}, \phi). \qquad \square$$

Given a program scheme $\mathscr{C}$, in practice the $\mathscr{D}$ required by Law 8.1 is found as before: $|[\textbf{inv} \ldots]|$ is distributed inwards until it reaches the recursive call p; then it is removed. What results is $\mathscr{D}$. An example is given in the next section.

8.3. Iteration

Iteration, written **do** $(\Box i \bullet G_i \to P_i)$ **od**, is just an abbreviation for this recursion:

mu p **•**
  if $(\Box i \bullet G_i \to P_i; p)$
  $\Box\ \neg(\bigvee i \bullet G_i) \to$ **skip**
  fi
um.

That completes Fig. 1 for Dijkstra's original language. Now by Laws 6.2, 6.1 and

6.4, we have for any context J and program X that the program

$$
\begin{array}{l}
|[\textbf{inv } I \bullet \\
\quad \textbf{if } (\Box i \bullet G_i \rightarrow P_i ; X) \\
\quad \Box \neg(\bigvee i \bullet G_i) \rightarrow \textbf{skip} \\
\quad \textbf{fi} \\
]|
\end{array}
$$

is refined under $\sqsubseteq_J$ by

$$
\begin{array}{l}
\textbf{if } (\Box i \bullet G'_i \rightarrow |[\textbf{inv } I \bullet P_i]| ; |[\textbf{inv } I \bullet X]|) \\
\Box \neg(\bigvee i \bullet G'_i) \rightarrow \textbf{skip} \\
\textbf{fi},
\end{array}
$$

provided that, for each i, $(J \wedge I) \Rrightarrow (G_i \Leftrightarrow G'_i)$. Law 8.1 then gives immediately:

Law 8.2 (Invariant distribution through iteration). *Providing, for each i, that* $(J \wedge I) \Rrightarrow (G_i \Leftrightarrow G'_i)$,

$$
\begin{array}{l}
\quad |[\textbf{inv } I \bullet \textbf{do } (\Box i \bullet G_i \rightarrow P_i) \textbf{ od}]| \\
\sqsubseteq_J \textbf{do } (\Box i \circ G'_i \rightarrow |[\textbf{inv } I \bullet P_i]|) \textbf{ od}
\end{array}
$$

9. Examples

Our examples are chosen to expose the difference in practice between explicit and implicit invariants: we present two developments of a program to calculate the greatest common divisor.

In the first example we use an explicit invariant; since it is not code, it must be removed "by hand" at the very end. During the development, however, it provides extra context. In the second example we use an implicit invariant instead, and during development we respect the type constraint it imposes. Then there is nothing significant to remove at the end.

Both developments require laws of refinement not already presented, and those have been placed in the appendix. Also required are logical constants and initial variables, which we now explain.

9.1. Logical constants and initial variables

Consider the following specification that x must increase:

$$\{X = x\}\, x : [x > X]. \tag{5}$$

It refines to $x := x+1$ for example—but it refines also to $x := X+1$. Usually the second refinement is not intended, since X is only a place-holder for the initial value of x and should not appear in the final program.

In fact, X above is a *logical constant*, which we now define precisely [11]. Logical constants are declared using **con**, and their scope indicated by a local block. This is the definition:

Definition 9.1 (Logical constant). Provided neither I nor ϕ contain free X,

$$wp_I(|[\mathbf{con}\ X \bullet P]|, \phi) \triangleq (\exists X \bullet wp_I(P, \phi)).$$

Logical constants need not be in upper case, of course, though for clarity we will follow that convention.

Definition 9.1 allows specification (5) to be rewritten

$$|[\mathbf{con}\ X \bullet \{X = x\}\ x:[x > X]]|, \tag{6}$$

and it can no longer be refined to $x := X + 1$.

Following are laws for introducing and removing logical constants.

Law 9.2 (Introduce logical constant).

$$P \sqsubseteq_I |[\mathbf{con}\ X \bullet P]|.$$

Note that Law 9.2 applies whether or not X is free in P; usually, however, the introduced logical constant is a fresh name.

Since a logical constant is not code, it must be removed after it has served its purpose; logical constants are used during development, but are not executed. They can be removed when all references to them have been eliminated:

Law 9.3 (Remove logical constant). *If X occurs nowhere in program P, then*

$$|[\mathbf{con}\ X \bullet P]| \sqsubseteq_I P.$$

There are many uses of **con**, and we shall see some in our examples to follow. But the most common use is to refer to initial values, and we introduce an abbreviation especially for that:

Abbreviation 9.4 (Initial variables). Occurrences of 0-subscripted variables in the postcondition of a specification refer to values held by those variables in the *initial* state. Let x be any variable, probably occurring in the frame w; if X is a fresh name, then

$$\begin{aligned} &\{pre\}\ w:[post] \\ \triangleq\ &|[\mathbf{con}\ X \bullet \{pre \wedge (X = x)\}\ w:[post[x_0 \backslash X]]]|. \end{aligned}$$

We reserve 0-subscripted names for that purpose, and call them *initial variables.*

Using Abbreviation 9.4, specification (6) is written $x:[x \geq x_0]$, since trivially we have $\{true\} = \mathbf{skip}$.

A more detailed discussion of logical constants and initial variables appears in [11].

9.2. First example: Explicit invariant

In the first example, we assume the two numbers whose gcd is to be found are both integers: that is, the development is carried out in the context of a declaration **var** $a, b : \mathbb{Z}$. The algorithm we derive assumes as well, however, that both are nonnegative initially, as reflected in its specification:

$$
\begin{array}{l}
\|[\textbf{con } G \bullet \\
\quad \{(a,b \geq 0) \wedge (G = \gcd(a, b))\}\ a, b : [a = G] \qquad \lhd \\
]\|.
\end{array}
$$

The sign $\lhd$ in the right margin indicates the part of the program we next refine; the surrounding text is unchanged.

We decide that we will maintain the condition $a,b \geq 0$ throughout, and by making it an invariant we avoid having to carry it explicitly through the development. We proceed (in small steps for illustration)

$$
\begin{array}{ll}
\sqsubseteq & \text{"Law A.1 (Split assumption)"} \\
 & \{a,b \geq 0\}\ \{G = \gcd(a, b)\}\ a, b : [a = G] \\
\sqsubseteq & \text{"Law 4.3 (Annotations and skip)"} \\
 & \{a,b \geq 0\}\ \{G = \gcd(a, b)\}\ a, b : [a = G]; [a,b \geq 0] \\
\sqsubseteq & \text{"Law 5.1 (Local invariant)"} \\
 & \|[\textbf{inv } a,b \geq 0 \bullet \\
 & \quad \{G = \gcd(a, b)\}\ a, b : [a = G] \qquad \lhd \\
 &]\|.
\end{array}
$$

Note that all the refinements above occur in the context $a,b \in \mathbb{Z}$ provided by the assumed declaration of a and b as integers. Thus the relation between successive lines is actually $\sqsubseteq_{a,b:\mathbb{Z}}$, though that would be tedious to write out each time. We assume therefore in setting out developments that the refinements are relative to all enclosing invariants.

Now we anticipate an iteration terminating with one of a and b equal to zero, and the other holding the gcd. So we make the following step, again refining only the part indicated by $\lhd$ above:

$$
\begin{array}{ll}
\sqsubseteq & \text{"Law A.2 (Following assignment)"} \\
 & \{G = \gcd(a, b)\}\ a, b : [a + b = G]; \qquad \lhd \\
 & a := a + b.
\end{array}
$$

The refinement relation this time (and from here on) is $\sqsubseteq_{a,b:\mathbb{N}}$, since we are now enclosed by the invariants $a,b \in \mathbb{Z}$ (implicit) and $a,b \geq 0$ (explicit).

The iteration is then introduced using Law A.3; the invariant and variant are included as comments in the development. Formulae stacked vertically are implicitly conjoined.

$\sqsubseteq$ "invariant: $G = \gcd(a, b)$; variant: $a + b$"

do $a \geq b > 0 \rightarrow$

$$\left\{\begin{matrix} a \geq b > 0 \\ G = \gcd(a, b) \end{matrix}\right\} a, b : \left[\begin{matrix} a + b < a_0 + b_0 \\ G = \gcd(a, b) \end{matrix}\right] \tag{7}$$

$\square$ $b \geq a > 0 \rightarrow$

$$\left\{\begin{matrix} b \geq a > 0 \\ G = \gcd(a, b) \end{matrix}\right\} a, b : \left[\begin{matrix} a + b < a_0 + b_0 \\ G = \gcd(a, b) \end{matrix}\right] \tag{8}$$

od.

Although Law A.3 bounds the variant below by 0, we leave that out in (7) and (8) since our context provides it. (More precisely, we use Law 4.2 to remove it.)

Now the two conditions required by Law A.3 for the above step are:

$$(a, b \in \mathbb{N}) \wedge (G = \gcd(a, b)) \Rrightarrow G = \gcd(a, b) \tag{9}$$

$$(a, b \in \mathbb{N}) \wedge \begin{pmatrix} G = \gcd(a, b) \\ (a < b) \vee (b \leq 0) \\ (b < a) \vee (a \leq 0) \end{pmatrix} \Rrightarrow a + b = G. \tag{10}$$

Condition (9) is obviously met, and (10) follows from this:

$$(a, b \in \mathbb{N}) \wedge \begin{pmatrix} G = \gcd(a, b) \\ (a = 0) \vee (b = 0) \end{pmatrix} \Rrightarrow a + b = G.$$

Finally, we complete the development by refining the guarded statements to assignments as follows:

(7)

$\sqsubseteq$ "Law 4.4 (Assignment)"

$a := a - b$

(8)

$\sqsubseteq$ "Law 4.4 (Assignment)"

$b := b - a$.

We leave out the assumptions (Law 4.3), and in the proviso assume additionally that $(a = a_0) \wedge (b = b_0)$. Strictly speaking, that is an extension of Law 4.4 to account for Abbreviation 9.4.

At this point the collected program is:

|[con G; **inv** $a, b \geq 0$ •

do $a \geq b > 0 \rightarrow a := a - b$

$\square$ $b \geq a > 0 \rightarrow b := b - a$

od;

$a := a + b$

]|.

Except for the declarations **con** and **inv**, we have reached code. And since G is not used in the program, the **con** is removed by Law 9.3.

The invariant $a,b \geq 0$ is not so easy to remove. We distribute it inwards, using Laws 6.1 and 8.2. That gives:

$$\begin{array}{l}\textbf{do } a \geq b > 0 \rightarrow |[\textbf{inv } a,b \geq 0 \bullet a := a - b]| \\ \Box\ b \geq a > 0 \rightarrow |[\textbf{inv } a,b \geq 0 \bullet b := b - a]| \\ \textbf{od}; \\ |[\textbf{inv } a,b \geq 0 \bullet a := a + b]|.\end{array}$$

Of the three inner invariant blocks, only the last can be immediately replaced by its body (using Law 6.3), because the associated condition for that removal is trivially true:

$$a,b \in \mathbb{N} \Rrightarrow a + b, b \geq 0.$$

But to use that law for the other invariant blocks requires the conditions

$$a,b \in \mathbb{N} \Rrightarrow a - b, b \geq 0,$$

$$a,b \in \mathbb{N} \Rrightarrow a, b - a \geq 0,$$

neither of which is true. In fact, elimination of those invariants requies an assumption which we have discarded—if in using Law A.3 we had retained (and weakened by Law 4.1) the loop guards as assumptions, we would have instead

$$\begin{array}{l}\textbf{do } a \geq b > 0 \rightarrow \\ \qquad |[\textbf{inv } a,b \geq 0 \bullet \{a \geq b > 0\}\ a := a - b]| \\ \Box\ b \geq a > 0 \rightarrow \\ \qquad |[\textbf{inv } a,b \geq 0 \bullet \{b \geq a > 0\}\ b := b - a]| \\ \textbf{od}; \\ a := a + b.\end{array}$$

Now the inner invariant blocks can be eliminated using Law A.4, giving the following program:

$$\begin{array}{l}\textbf{do } a \geq b > 0 \rightarrow a := a - b \\ \Box\ b \geq a > 0 \rightarrow b := b - a \\ \textbf{od}; \\ a := a + b.\end{array}$$

The automatic type-checking finally required by the original declaration **var** $a, b : \mathbb{Z}$ is trivial, since $\mathbb{Z}$ is closed under both addition and subtraction. That completes the development.

Notice that the program fragment (7) could have been refined even to the ridiculous $a := a - 5b$, say, but would never then have led to code. In removing the invariant, we would have reached

$$|[\textbf{inv } a,b \geq 0 \bullet \{a \geq b > 0\}\ a := a - 5b]|.$$

And there we would have remained, as we cannot satisfy the proviso

$$(a,b \in \mathbb{N}) \wedge (a \geq b > 0) \Rrightarrow a - 5b, b \geq 0.$$

9.3. Second example: Implicit invariant

In the second example, we assume the two numbers whose gcd is to be found are both natural numbers: that is, that the development is carried out in the context of a more restrictive declaration **var** $a, b : \mathbb{N}$. There is no need to record an explicit assumption that a and b are nonnegative, therefore, since that is known from their type. We begin with

$$
\begin{array}{l}
|[\textbf{con } G \bullet \\
\quad \{G = \gcd(a, b)\}\ a, b : [a = G] \qquad \triangleleft \\
]|.
\end{array}
$$

As before, we introduce the final assignment and the loop:

$$
\begin{array}{l}
\sqsubseteq \{G = \gcd(a, b)\}\ a, b : [a + b = G]; \qquad \triangleleft \\
\quad a := a + b \\
\\
\sqsubseteq \quad \text{"invariant: } G = \gcd(a, b)\text{; variant: } a + b\text{"} \\
\quad \textbf{do } a \geq b > 0 \rightarrow \\
\qquad \left\{ \begin{matrix} a \geq b > 0 \\ G = \gcd(a, b) \end{matrix} \right\} a, b : \left[\begin{matrix} a + b < a_0 + b_0 \\ G = \gcd(a, b) \end{matrix} \right] \qquad (11) \\
\\
\quad \square\ b \geq a > 0 \rightarrow \\
\qquad \left\{ \begin{matrix} b \geq a > 0 \\ G = \gcd(a, b) \end{matrix} \right\} a, b : \left[\begin{matrix} a + b < a_0 + b_0 \\ G = \gcd(a, b) \end{matrix} \right] \qquad (12) \\
\quad \textbf{od}.
\end{array}
$$

Note the simplification again of $0 \leq a + b < a_0 + b_0$, this time justified because it is in the scope of the implicit invariant $a, b \in \mathbb{N}$.

The guarded statements are refined by assignments, but we use natural number subtraction so that the subsequent type-checking will succeed:

$$(11) \sqsubseteq a := a \ominus b,$$

$$(12) \sqsubseteq b := b \ominus a.$$

It would have been correct at this point to use ordinary subtraction $a - b$ as before, but that would fail the type-checking later imposed by **var** $a, b : \mathbb{N}$. It would have been correct, too, in the first example to use $a \ominus b$; then the removal of the explicit invariant would not have required the re-introduced assumptions.

Collecting the program at this point reveals

$$
\begin{array}{l}
|[\textbf{con } G \bullet \\
\quad \textbf{do } a \geq b > 0 \rightarrow a := a \ominus b \\
\quad \square\ b \geq a > 0 \rightarrow b := b \ominus a \\
\quad \textbf{od}; \\
\quad a := a + b \\
]|.
\end{array}
$$

The logical constant G is removed as before, leaving the final program

$$\begin{array}{l}\textbf{do } a \geqslant b > 0 \rightarrow a := a \ominus b \\ \Box\ b \geqslant a > 0 \rightarrow b := b \ominus a \\ \textbf{od}; \\ a := a + b.\end{array}$$

Now the type-checking (described in Section 7) of the above code can be performed automatically, and succeeds since $\mathbb{N}$ is closed under $\ominus$. But $\mathbb{N}$ is not closed under ordinary subtraction, so using $a - b$ would have caused type-checking to fail.

Our two examples show that an implicit invariant is more convenient than the equivalent explicit one, since then nothing need be explicitly removed. The type-checking is automatic, and that corresponds to ordinary programming practice. Explicit invariants, however, allow the rigorous use of invariants that cannot be decidably type-checked. The benefit is that they, like types, can be assumed everywhere and need not be copied from place to place; the price is the explicit reasoning, finally, to remove them.

10. A discussion of motives

The definition of $wp_I(\cdot, \cdot)$ was suggested by a certain kind of data refinement. Let $P \leqslant P'$ mean that P is data-refined to P' under the transformation

(no abstract variables, no concrete variables, coupling invariant I).

Such data refinements are described in [7]. From the definitions there, we have that if $P \leqslant P'$ then for all ϕ,

$$I \wedge wp(P, \phi) \Rightarrow wp(P', I \wedge \phi).$$

It can be shown that the least-refined such P' is defined by

$$wp(P', \phi) \ = \ I \wedge wp(P, I \Rightarrow \phi), \tag{13}$$

and that is the motivation for Lemma 5.2. (Incidentally, there is a corresponding formula for data refinements in general: it is

$$(\exists a \bullet I \wedge wp(P, (\forall c \bullet I \Rightarrow \phi))).$$

We have just taken the special case in which the lists a and c of abstract and concrete variables are both empty.)

Equation (13) is where the definitions of Fig. 1 and Section 3.1 come from, and it is the uniform application of that which gives an invariant-breaking assignment its miraculous semantics, just as ill-advised data refinements lead to miracles [8].

We know that data refinement has nice distribution properties, and that is why $|[\textbf{inv } I \bullet \ldots]|$ does too. One can see

$$|[\textbf{inv } I \bullet P]|$$

as the program got by distributing the data refinement, above, through P. And that is why Lemma 5.2 is true: such distribution can only refine the program.

11. Related work

The refinement calculus, first proposed by Back [1], has in fact been invented twice more [9, 14]. At Oxford it was made specifically for the rigorous development of programs from Z specifications [4, 18].

A prominent feature of Z specifications is the *schema* which, when used to describe abstract operations, carries an invariant around with it that includes type information and is maintained automatically. That is where we started.

Where we have finished is very similar to work by Lamport and Schneider [6]. Their **constraints** clause and our **and** declaration have the same effect; their *constraint strengthening rule* is like our Law 5.1. Lamport and Schneider use *partial* correctness, however, and do not write specifications within their programs. Their constructions are defined within temporal logic; ours are defined by weakest preconditions.

Lamport and Schneider's use of partial correctness identifies aborting and miraculous behaviour, leading them to say that invariant-breaking assignments abort. For them, such programs establish *false* if they terminate—therefore they do not terminate. Ours "damn the torpedoes" and terminate anyway. Section 12 explains why.

Within our refinement calculus, Lamport and Schneider's constraint $x \perp y$—"x and y are independent"—would be expressed instead as a dependency (their **may alias**). That dependency would in Definition 3.1 link the variables in the frame to the bound variables in its meaning, allowing a more general relationship than our present equality. The frame would be expanded, according to the aliasing dependencies, before being applied as a universal quantification. Finally, their proposed extension to generalised assignments $exp := exp'$ is already neatly done with specifications.

Invariants are used also by Reynolds [17], called *general invariants*, and are true from their point of occurrence to the end of the smallest enclosing block. But the *specification logic* does not give them a meaning, nor are they connected with type information. The temporary falsification of a general invariant is allowed, however, and we have not discussed that here: there are several approaches to pursue. Like Lamport and Schneider, Reynolds uses partial correctness.

12. Conclusions

Although the traditional $wp(\cdot, \cdot)$ is our $wp_{true}(\cdot, \cdot)$, the traditional $\sqsubseteq$, as in [1, 5, 9, 14], is *not* our $\sqsubseteq$. That is because Definition 3.7 insists that the implication hold for all contexts.

What have we lost? Not much. Here is an example; with our definition of refinement,

$$n := -1; n := 0 \not\sqsubseteq n := 0.$$

Just take the context $n \in \mathbb{N}$, and the left-hand side becomes miraculous: it cannot refine to code. But most refinement laws have *atomic* left-hand sides—after all, they are there to introduce structure, not remove it. And when the left-hand side is atomic, we do preserve the traditional refinement relation—because that is true for any data-refinement [1, 3, 15]. Thus most existing refinement laws remain valid. For example (the left-hand side is atomic), we still have

$$n := 0 \sqsubseteq n := -1; n := 0.$$

But the price of the miraculous assignment, when it appears to break the invariant, is the final type-checking. The "obvious" alternative is the operationally-motivated alternative definition

$$wp_t(x := E, \phi) \;\hat{=}\; I \wedge (I \wedge \phi)[x \backslash E].$$

All that does, however, is add the type-checking to Law 4.4, and we lose, *formally* at least, the ability to delay such checking until the final test for feasibility. More significant, however, is that using the above definition would not allow the refinement

$$x : [x = E] \sqsubseteq x := E.$$

(We assume that E contains no x.) For if that refinement were valid, then by Lemma 3.10 it would be valid in the context $x \neq E$ as well. But it cannot be: in that context, the left-hand side is a miracle but, with the alternative definition of assignment, the right-hand side would abort.

The ability to factor "details" like feasibility, and hence type-checking, is essential in a practical method. Experienced programmers will only be impeded by continual checking of types: their programs tend to be well-typed anyway. And in cases of error, the compiler acts at the last minute, catching the mistake. Because they are experienced, that will happen rarely.

Inexperienced programmers, however, will waste a lot of time by leaving their feasibility and type-checks till later. When eventually the check is made, and fails, all the intervening development must be discarded. Because they are inexperienced, that will happen often.

But we cannot exploit experience by allowing those having it to apply "only some of the rules". All programmers, experienced or not, must apply all of the rules; but for each, the set of rules to which "all" applies may be different. The experts' rules are those from which all feasibility and type checks have been removed; the rules for apprentices have all the checks built in, and performance suffers. A mathematical factorisation is necessary to make that distinction reliably; the one we have chosen is close to what people do already.

Appendix A. Additional refinement laws

These laws are used in Section 9.

Law A.1 (Split assumption).

$$\{pre \wedge pre'\} \sqsubseteq_I \{pre\}\{pre'\}.$$

Law A.2 (Following assignment). *For any term E,*

$$\begin{array}{l} w, x:[post] \\ \sqsubseteq_I\ w, x:[post[x\backslash E]]; \\ \quad x := E. \end{array}$$

Law A.3 (Loop introduction). *Providing $(I \wedge pre) \Rrightarrow inv$ and $(I \wedge inv \wedge \neg(\bigvee G_i)) \Rrightarrow post$, and v is any integer-valued expression,*

$$\begin{array}{l} \{pre\}\ w:[post] \\ \sqsubseteq_I\ \textbf{do}\ (\Box i \bullet G_i \rightarrow \{G_i \wedge inv\}\ w:[inv \wedge 0 \leq v < v_0])\ \textbf{od}. \end{array}$$

Law A.4 (Invariant elimination from preconditioned assignment). *Provided $(J \wedge I \wedge pre) \Rrightarrow I[w\backslash E]$,*

$$|[\textbf{inv}\ I \bullet \{pre\}\ w := E]| \sqsubseteq_I w := E.$$

Acknowledgement

We thank Paul Gardiner for his help, and Ian Hayes for his comments on the earlier version [10].

References

[1] R.-J.R. Back, Correctness preserving program refinements: Proof theory and applications, Tract 131, Mathematisch Centrum, Amsterdam (1980).
[2] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[3] P.H.B. Gardiner and C.C. Morgan, Data refinement of predicate transformers, *Theoret. Comput. Sci.* (to appear); reprinted in [13].
[4] I.J. Hayes, *Specification Case Studies* (Prentice-Hall, London, 1987).
[5] E.C.R. Hehner, *The Logic of Programming* (Prentice-Hall, London, 1984).
[6] L. Lamport and F.B. Schneider, Constraints: A uniform approach to aliasing and typing, in: *Proceedings Twelfth Symposium on Principles of Programming Languages*, New Orleans, LA (1985) 205–216.
[7] C.C. Morgan, Auxiliary variables in data refinement, *Inform. Process. Lett.* **29** (1988) 293–296; reprinted in [13].
[8] C.C. Morgan, Data refinement using miracles, *Inform. Process. Lett.* **26** (1988) 243–246; reprinted in [13].
[9] C.C. Morgan, The specification statement, *ACM Trans. Programming Languages Syst.* **10** (3) (1988); reprinted in [13].

[10] C.C. Morgan, Types and invariants in the refinement calculus, in: J.L.A. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science **375** (Springer, Berlin, 1989).

[11] C.C. Morgan, *Programming from Specifications* (Prentice-Hall, London, 1990).

[12] C.C. Morgan and K.A. Robinson, Specification statements and refinement, *IBM J. Res. Develop.* **31** (5) (1987); reprinted in [13].

[13] C.C. Morgan, K.A. Robinson and P.H.B. Gardiner, On the refinement calculus, Tech. Rept. PRG-70, Programming Research Group, Oxford (1988).

[14] J.M. Morris, A theoretical basis for stepwise refinement and the programming calculus, *Sci. Comput. Programming* **9** (1987) 287-306.

[15] J.M. Morris, Laws of data refinement, *Acta Inform.* **26** (1989) 287-308.

[16] G. Nelson, A generalization of Dijkstra's calculus, *ACM Trans. Programming Languages Syst.* **11** (1989) 517-561.

[17] J.C. Reynolds, *The Craft of Programming* (Prentice-Hall, London, 1981).

[18] J.M. Spivey, *The Z Notation: A Reference Manual* (Prentice-Hall, London, 1989).

[19] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).